OLA Testing Flow

Motability Operations

Agenda

Breakdown

- Introduction and Testing Strategy
- Visualising the flow
- Testing: Initial Form/Application
- Testing: DVLA & Eligibility Checks
- Mocking and Edge Cases
- Test Reporting

Introduction & Testing Strategy

Visualising the flow

Visualisation of the entire flow helps to see the journey from POST API call to final submission after third-party checks

Breaking up the flow

Once visualised, identifying key stages to test and where integration and end to end testing should be layered becomes clearer for each section of the flow

Testing Principles

For each section/feature of the flow, we should consider the positive, negative and edge cases, and use of mocking

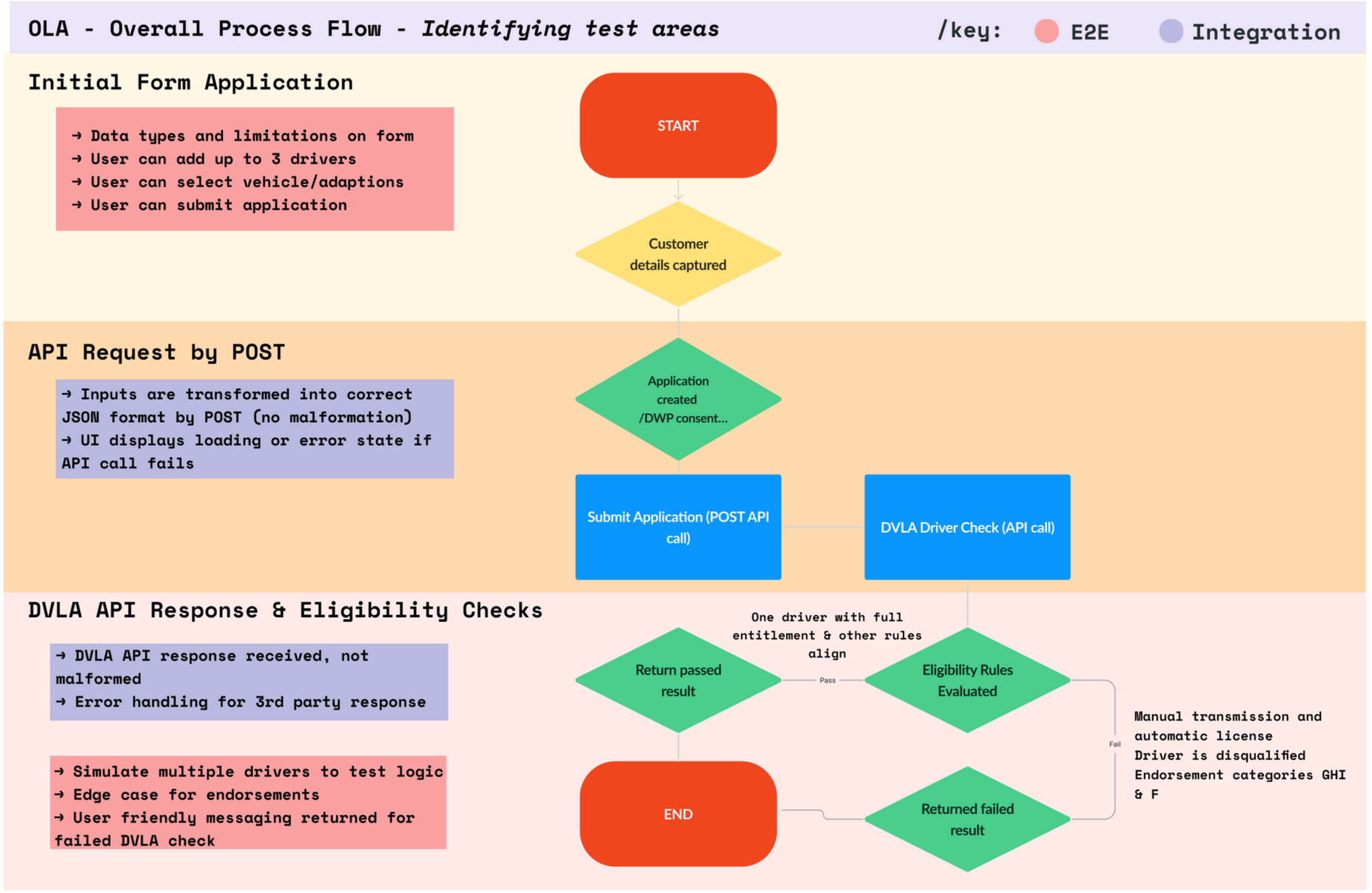
Eligibility Checks

Verifying that eligibility rules and DVLA checks responds correctly in a variety of scenarios

Test Reporting

Reporting on the passes, fails of the flow and assigning a level of urgency for Developers to work on fixes

Visualising the flow



- This is the flow of the API call from application submission, to the third-party call to the DVLA
- Break flow down into 3 core sections to test
  - Initial Application
  - API POST Request
  - DVLA and Eligibility Checks
- Test cases to be built up:
  - 1) Integration with API request/response
  - 2) Simulating real world scenarios of OLA software use
- Unit testing would happen during development, so I have focused on testing that would happen at this stage - though as a tester, you may review test coverage with developers!

Testing: Initial Form/Application

See full feature tests here

```
Feature: Initial Form Application

  As a dealer submitting a customer's vehicle application,
  I want to ensure the form handles data types, validations, and submissions correctly
  So that I can submit a valid application to the system

  # ====================
  # Happy Path Tests
  # ====================

  @e2e-testing

  Scenario: User enters valid form data and submits the application
    Given the user enters valid customer details (first name, last name, etc.)
    And the user adds 2 additional drivers with valid details
    And the user selects a valid vehicle and adaptations
    And the user selects a valid estimated delivery date
    And the user confirms the communication preferences
    When the user submits the application
    Then the system should submit the application successfully
    And the application should be saved and processed by the system

  # ====================
  # Negative Tests
  # ====================

  @e2e-testing

  Scenario: User adds more than 3 drivers
    Given the user adds 4 drivers to the application
    When the user attempts to submit the form
    Then the system should show an error message: "You can only add up to 3 drivers"
```

- I've broken up test scripts using Gherkin case to show how I would test each part of the flow, covering Integration where applicable, and end to end testing. This is good syntax to eventually automate from.
- Considered positive testing, negative testing, and edge cases

- For this stage, we need to make sure that the form data is submitted correctly for the POST request to be successful
- Ensure a user/dealer fills in all necessary information to process the check
- Consideration to what will show up to the user when information is missing to be able to proceed
- Edge Case: what if we submit the same driver details? What if a named driver has details missing?

Testing: API POST Request

See full feature tests here

```
Feature: API Request by POST

  As a dealer submitting vehicle eligibility requests,
  I want to ensure the system formats and sends the correct API request
  So that the eligibility data can be processed correctly by the system

  # ====================
  # Happy Path Tests
  # ====================

  @integration-testing

  Scenario: Inputs are transformed into the correct JSON format by POST
    Given the user has entered valid form data
    When the user submits the application
    Then the system should send a POST request to "/api/v2/applications" with the correct JSON body
    And the JSON should include customer details, drivers, vehicle adaptations, etc.

  # ====================
  # Negative Tests
  # ====================

  @integration-testing

  Scenario: UI displays loading or error state if API call fails
    Given the user submits the application
    When the API call fails (e.g., due to network issues)
    Then the UI should display an error message "Unable to submit application. Please try again later."
    And the user should see a loading spinner while the request is being processed
```

- I've broken up test scripts using Gherkin case to show how I would test each part of the flow, covering Integration where applicable, and end to end testing. This is good syntax to eventually automate from.
- Considered positive testing, negative testing, and edge cases
- Testing that the API request is being sent correctly before it reaches the next stage DVLA check
- Ensure the JSON data is not malformed via POST
- Ensure that the user can see any errors, if any, during the request
- Edge Case: what if the submission is double-clicked and triggers more than one request?

Testing: DVLA & Eligibility Checks

See full feature tests here

```
Feature: DVLA API Response and Eligibility Checks

  As a dealer receiving eligibility data from the DVLA API,
  I want to ensure the system processes the response correctly and applies business rules
  So that I can confirm the eligibility of the vehicle and driver(s)

  # ====================
  # Happy Path Tests
  # ====================

  @integration-testing

  Scenario: DVLA API response received and parsed correctly
    Given the system submits driver details to the DVLA API
    When the system receives a response with valid eligibility data
    Then the system should process the response and apply business rules correctly
    And the eligibility status should be displayed to the user

  @e2e-testing

  Scenario: Vehicle meets eligibility criteria
    Given the DVLA API returns a response with "eligible: true"
    When the system processes the response
    Then the system should display "Vehicle is eligible"
    And proceed to the next step in the application


  # ====================
  # Negative Tests
  # ====================

  @integration-testing

  Scenario: API response returns an error status
    Given the DVLA API returns a 500 error response
```

- I've broken up test scripts using Gherkin case to show how I would test each part of the flow, covering Integration where applicable, and end to end testing. This is good syntax to eventually automate from.
- Considered positive testing, negative testing, and edge cases
- Eligibility logic is core focus at this stage, where integration and end to end testing validate business critical decisions
- We ensure that the API response is parsed correctly, and rules are applied correctly
- This stage is where we determine if someone qualifies and needs to be tested thoroughly
- Wide range of scenarios are covered, including both eligible and ineligible drivers, edge cases
- Mocking/simulating multiple requests would be useful to test the combinations of endorsements
- Checking API response alongside each scenario will help to see if different combinations fail, pass and where

- Edge Case: what if owner has different licence types, across different vehicle categories?

Mocking and Edge Cases

Crucial to building all round test coverage

Mock APIs

Introduction

Web APIs are usually implemented as HTTP endpoints. Playwright provides APIs to **mock** and **modify** network traffic, both HTTP and HTTPS. Any requests that a page does, including XHRs and fetch requests, can be tracked, modified and mocked. With Playwright you can also mock using HAR files that contain multiple network requests made by the page.

Mock API requests

The following code will intercept all the calls to `*/**/api/v1/fruits` and will return a custom response instead. No requests to the API will be made. The test goes to the URL that uses the mocked route and asserts that mock data is present on the page.

```
test("mocks a fruit and doesn't call api", async ({ page }) => {
  // Mock the api call before navigating
  await page.route('*/**/api/v1/fruits', async route => {
    const json = [{ name: 'Strawberry', id: 21 }];
    await route.fulfill({ json });
  });
  // Go to the page
  await page.goto('https://demo.playwright.dev/api-mocking');

  // Assert that the Strawberry fruit is visible
  await expect(page.getByText('Strawberry')).toBeVisible();
});
```

- When actually coding the test scripts, we should consider ‘mocking’ to test areas in isolation - helps to see that tests are not dependent on external services
- Keeps tests reliable even if the external API is down or is not behaving as predicted
- Speed up performance and eliminate need for real Network calls
- Edge cases are useful throughout for seeing how the software behaves in unusual or unexpected outcomes
- Helps to prevent unexpected failures, bugs and improves stability and keeps system robust!

Test Reporting

Essential to all stakeholders and developers

- Once tests have been run, providing a test report will make the results clear and visible to wider team
- Helps to collectively identify what is functioning as expected and where issues need to be addressed
- Quality assurance from insights and test coverage
- Identify trends if tests are run over time
- Track bugs and issues, by grading these in urgency, we can work with developers on a method of prioritising issue fixes
- Inclusion of information as to why tests failed, with error messaging and logs, developers can address more easily
- Helps to document the entire process and keep track of what has been done

Thank you!

Please let me know if you have any questions!